Type-Safe Homogeneous Linkage For Heterogeneous Smart Pointers

Martin D. Richek

Related Applications

[0001] This applications claims the benefit of U.S. Provisional Application Serial No. 60/269,232, filed on February 16, 2001 and U.S. Provisional Application Serial No. 60/271,526 filed on February 26, 2001, the entire contents of which are incorporated herein by reference.

Field of the Invention

[0002] The present invention relates generally to a computer method for managing the lifetime of a memory-resident element, and more specifically to a framework for use in an object-oriented environment that provides type-safe homogeneous linkage for heterogeneous smart pointers.

Background of the Invention

[0003] A frequently encountered problem in object-oriented programming is the control of the lifetime of dynamically allocated objects. This problem shows up in two forms. The first is memory leak, where the memory assigned to dynamically allocated objects is not released to the system after that memory is no longer being used. Such failure to release unused memory creates hardware and software inefficiencies and can require more memory than otherwise needed, thus needlessly increasing hardware cost. More significantly, in more complex software applications where a multiplicity of objects are dynamically allocated, the failure to release the associated memory when the object is no longer needed will often result in run-time errors, i.e., program crashes.

[0004] The second problem is a dangling pointer, where reference is made to an object after its memory has been released to the system (i.e., the object is no longer available). Where a reference is made to a dangling pointer the run-time result is unpredictable and may result in an invalid value being returned, crashing of the program, or both.

[0005] Since many object-oriented programming languages, such as C++, do not inherently release unused memory or check for a dangling pointer condition, an added burden is imposed on the programmer to provide code to keep track of memory release and object lifetime. Adding such code to monitor these conditions produces a number of undesirable effects. First, providing additional code increases the program development cost, since the programmer must dedicate time to addressing such problems. As programs become larger and more complex, it also becomes more difficult to comprehensively and reliably provide code to keep track of memory release and eliminate the possibility of a dangling pointer. For example, in particularly large programs, multiple programmers may work on different code sections, in which case providing such additional code consistently and uniformly throughout the program can be difficult to achieve, and therefore can add significant development cost.

[0006] Second, the manner in which the additional code is implemented has the potential for decreasing the performance of the resulting program, as processor time must be spent executing the additional code. In addition, the additional code may necessitate added hardware, e.g., memory. Hence, there remains a need to provide program developers with programming tools that afford a convenient and efficient approach to control and monitor memory release and object lifetimes.

[0007] In addition, many software applications are already written in languages such as C++, including major commercial products such as database management programs, where the above problems have not been adequately addressed. Failure to address the problems yields software applications that run less efficiently and reliably than is otherwise possible. In many instances, the existing software applications which suffer from these problems are ones that are particularly complex and contain many lines of code. Consequently, an intrusive method of monitoring the memory usage is undesirable as it requires rewriting major portions of lengthy, complex code. Thus, there also remains a need for non-intrusive, retrofit programming tools that provide for efficient memory allocation and minimize performance cost. Moreover, in many instances, memory allocation relates to a number of pointers some of which may point not only to a particular object, but also to differently-typed sub-objects of the object. The pointers to sub-objects, like pointers to the object containing the sub-objects, also need to be monitored to avoid memory release problems and the dangling pointer condition. In addition, such monitoring needs to be maintained as pointers to differently-typed sub-objects are assigned to one another. Thus, there is a need for programming tools which also enable management of heterogeneous pointers.

Summary of the Invention

[0008] In accordance with the present invention, a computer-implemented method of memory management is provided. The method includes the step of providing a linked list comprising at least one smart pointer associated with a memory-resident element. As used herein, a smart pointer means a user-defined type that includes the behavior of a built-in pointer and adds additional functions. The linked list may take the form of a bi-directional, doubly-linked ring. The smart pointer includes a next pointer for pointing to a selected smart pointer, which may be a different smart pointer on the linked list or may be the selected smart pointer itself. The method includes the step of comparing the value of the memory location of a selected smart pointer giving up its association with the memory-resident element to the value of the next pointer of the selected smart pointer, to provide a determination whether the linked list contains only the selected smart pointer. If the value of the memory location of the selected smart pointer equals the value of the next pointer of the selected smart pointer, then the linked list comprises only the selected smart pointer. The method also provides the step of deleting the memory-resident element when the value of the memory location of a selected smart pointer equals the value of the next pointer of the selected smart pointer, whereby the memory assigned to the memory-resident element is released when no further reference can be made to the memory-resident element. In order to provide type-safety, the step of providing a common base to all types of smart pointers is provided by the method of the invention. The method includes the step of providing a function that provides automatic conversion between smart pointers of different classes in the same class hierarchy of the common base.

[0009] A memory-resident structure of an object-oriented programming environment arranged to manage the life-time of an object is also provided. The memory-resident structure includes a linked list comprising at least one smart pointer for keeping track of references to an object. The smart pointer includes a next pointer for pointing to a smart pointer on the linked list. The memory-resident structure also includes a function for deleting the object when the value of a selected next pointer equals the value of the memory location of the smart pointer in which the selected next pointer is included. The memory-resident structure includes a function that provides automatic conversion between smart pointers of different classes in the same class hierarchy.

Brief Description of the Drawings

[0010] The foregoing summary and the following detailed description of the preferred embodiments of the present invention will be best understood when read in conjunction with the appended drawings, in which:

[0011] Figure 1 schematically illustrates a bi-directional ring of smart pointers having two members;

[0012] Figure 2 schematically illustrates the ring of Figure 1 with an additional smart pointer inserted into the ring;

[0013] Figure 3 schematically illustrates the ring of Figure 1 with one of the members removed from the ring; and

[0014] Figure 4 schematically illustrates a flowchart of a function for managing a smart pointer which is giving up its object which includes a test for the condition that a ring has only a single-member.

Detailed Description of the Invention

[0015] One aspect of the invention relates to a linked list or ring of non-intrusive smart pointers each of which is associated with a particular memory-resident element. One or more linked lists or rings may be active at the same time during the running of a program, however each list or ring contains all smart pointers associated with a particular memory-resident element, including pointers that point directly to the element or to a sub-element of the element. The linked list or ring provides a structure which can be monitored to determine when the last smart pointer to a particular pointed-to memory-resident element expires, whereupon the element itself may be deleted to release the memory allocated to the element. Use of a linked list or ring provides non-intrusive monitoring of a memory-resident element without the need for extra memory allocation and release to provide the monitoring. The non-intrusive aspect of the smart pointer list of the present invention permits use of the smart pointer list in existing applications without intrusive revision of the application code.

[0016] The linked list may be provided as either a singly-linked or doubly-linked list or ring. While a linear list may be used, a ring list can provide a performance improvement over a linear list by eliminating the need to test for the end-of-list condition. Thus, referring to Fig. 1, one specific embodiment of the present invention for use in an object-oriented environment provides a bi-directional, doubly-linked ring 100 to monitor smart pointers to a memory-resident object 30.

[0017] The ring 100 comprises one or more smart pointers, and in the particular case illustrated in Fig. 1, the ring 100 contains a first smart pointer 10 and a second smart pointer 20. Each smart pointer 10, 20 includes an object pointer 12, 22 that points to the same object 30. In order for a smart pointer to be a member of the ring 100, the smart pointer must point to the same object 30 or to a sub-object of that object 30. The first smart pointer 10, second smart pointer 20, and the pointed-to object 30 each has a respective memory address 11, 21, 31 associated therewith. In the case where the first and second smart pointers 10, 20 both point to the same object 30, the object pointers 12, 22 both contain the memory address 31 of the object 30, which, by way of example, has a value of 0003 as shown in Fig. 1.

[0018] Each smart pointer 10, 20 includes a "Next" pointer 16, 26 and "Previous" pointer 14, 24 for pointing to the next member of the ring 100 and the previous member of the ring 100, respectively. In particular, as shown in Fig. 1, the first smart pointer 10 includes a "Next" pointer 16 which points to the second smart pointer 20 and includes a "Previous" pointer 14 which points to the second smart pointer 20. In the case shown where there are only two smart pointers on the ring, the second smart pointer 20 is both the next member on the ring 100 after the first smart pointer 10 and is also the previous member on the ring 100 before the first smart pointer 10. Hence, the "Next" pointer 16 and the "Previous" pointer 14 of the first smart pointer 10 each contain the memory address 21 of the second smart pointer 20, which, by way of example, has a value of 0002 as shown in Fig. 1. In a similar fashion, the second smart pointer 20 includes a "Next" pointer 26 which points to the first smart pointer 10 and includes a "Previous" pointer 24 which points to the first smart pointer 10. Thus, the "Next" pointer 26 and the "Previous" pointer 24 of the second smart pointer 20 each contain the memory address 11 of the first smart pointer 10, which has an exemplary a value of 0001 as shown in Fig. 1. In the case where more than two smart pointers are present on the ring, the "Next" pointer and the "Previous" pointer of a particular smart pointer will point to different smart pointers on the ring.

[0019] For example, with reference to Fig. 2, a three-member ring 100 is shown which is created by inserting a third smart pointer 40 between the first smart pointer 10 and the second smart pointer 20 of the ring 100 of Fig. 1. A smart pointer is added to the ring 100 when a smart pointer associated with the same object 30 is assigned or copied. All smart pointers associated with a particular object are located on the same ring to provide complete monitoring of all pointers to the particular object. Like the first and second smart pointers 10, 20, the third smart pointer 40 also contains an object pointer 42 that points to the object 30 or to a sub-object of object 30. To insert the third smart pointer 40 between the first smart pointer 10 and the second smart pointer 20, the "Next" pointer 16 of the first smart pointer 10 and the "Previous" pointer 24 of the second smart pointer 20 are each changed to point to the third smart pointer 40. Thus, the "Next" pointer 16 of the first smart pointer 10 and the "Previous" pointer 24 of the second smart pointer 20 each contain the memory address 41 of the third smart pointer 40, which, by way of example, has a value of 0004 as shown in Fig. 2. The third smart pointer 40 includes a "Next" pointer 46 which points to the second smart pointer 20 and includes a "Previous" pointer 44 that points to the first smart pointer 10. In a C++ embodiment of the invention, as provided in the Example below, the insertion operation of a smart pointer onto a ring is provided by the member functions ***void Attach (const PtrBase & other) const*** and ***void AttachX (const PtrBase & other) const***. ***Attach*** and ***AttachX*** perform the same function from opposite frames of reference. ***Attach*** attaches ***this PtrBase*** to the ring of which the ***other PtrBase*** is a member. As used herein with reference to the C++ embodiment of the Example, the term "***this***" refers to a hidden pointer passed to each member function that points to the object for which the member function has been invoked. ***AttachX*** attaches the ***other PtrBase*** to the ring of which ***this PtrBase*** is a member. Since ***PtrBase*** is a base class that may be in use by different derived classes, the presence of both member functions allows ring insertion when one derived class may not have access to the ***PtrBase*** member functions of the other.

[0020] When a smart pointer gives up its object pointer — whether it is receiving a new object pointer or it is expiring— then, that smart pointer is removed from the ring of which it is a member. In particular, to know when an object is no longer needed and the object's memory should therefore be released, it is important to test whether the smart pointer giving up its object pointer is the last member of the ring. If the smart pointer is the last member of the ring, the object to which it points may be deleted. The "last member of the ring" test is illustrated in Fig. 4.

[0021] A smart pointer that is alone on a ring points to itself as both the next and the previous "members" of the ring, as illustrated in Fig. 3. That is, the "Next" pointer 16 and "Previous" pointer 14 of the first smart pointer 10 each point to the first smart pointer 10. Thus, the values of the "Next" pointer 16 and "Previous" pointer 14 are equal, and each contains the memory address of the smart pointer 10 to which it belongs. More importantly, a significant distinction exists between the "Next" and "Previous" pointer values of each smart pointer in the single-member ring 100 of Fig. 3 and the two-member ring 100 of Fig. 1. In the two-member ring 100, the "Next" pointer 16 and the "Previous" pointer of the each smart pointer contain the memory address of the other smart pointer. In contrast, for the single-member ring 100, the "Next" pointer 16 and the "Previous" pointer 14 of the first smart pointer 10 each contains the memory address of the smart pointer to which it belongs. Thus, while the values of the "Next" pointer 16 and the "Previous" pointer 14 are equal to each other in both the single-member ring and the two-member ring, the values of the "Next" pointer 16 and the "Previous" pointer 14 are equal to the address of the smart pointer to which they belong only when the ring 100 contains a single-member as shown in Fig. 3. Therefore, the test for a single-member ring, step 410 of Fig. 4, comprises comparing the value of either the "Next" pointer 16 or the "Previous" pointer 14 to the address of the smart pointer to which the "Next" pointer 16 and "Previous" pointer 14 belong, e.g., the first smart pointer 10. If the test for equality between the value of the "Next" pointer 16 and the address of the first smart pointer 10, as provided at step 410, is true, then the object 30 is deleted at step 420. In the C++ embodiment of the invention of the Example below, the test of step 410 is provided by the member function ***bool IsOnly () const***.

[0022] If the test of step 410 returns "false", then the ring contains more than one smart pointer, and the smart pointer which is giving up its object pointer is removed from the ring. For example, if the ring contains three elements as shown in Fig. 2 and the smart pointer which is giving up its object pointer is the third smart pointer 40, then the third smart pointer 40 is removed from the ring 100 yielding the two-member ring shown in Fig. 1. The "Next" pointer 16 of the first pointer 10 and "Previous" pointer 24 of the second smart pointer 20 are set to point to the second smart pointer 20 and the first smart pointer 10, respectively, as explained above with reference to Fig. 1. In the C++ embodiment of the invention of the

Example below, detaching a smart pointer from a ring is provided by the member function ***void Detach () const.***

[0023] A further aspect of the present invention relates to a framework for providing type-safe homogeneous linkage for heterogeneous smart pointers. The framework comprises a base common to all smart pointers and a template for managing inter-class assignment and inter-class conversion of smart pointers. All ring operations, such as those discussed above, are contained in the common base shared by all smart pointers. Providing a base common to all smart pointers is desirable in class-hierarchical programming environments, such as C++, where the derived pointer is typed, since provision of a common pointer base allows pointers to objects of different classes in a common class hierarchy that point to different sub-objects of the same object to be members of a single ring, providing a complete representation of all references to an object. Alternatively, in a language that does not allow a pointer to be typed, the pointer base can be part of the pointer rather than its base. One embodiment of the common base is provided in the Example below. Although the Example is written in C++ and is therefore discussed using C++ terminology, the subject matter and underlying concepts apply equally well to other object-oriented programming languages.

[0024] Referring now to the Example, the common base defines the attributes of a smart pointer which may be used in the bi-directional, doubly-linked ring embodiment disclosed above in reference to Figs. 1-4. The attributes of the common base, ***PtrBase***, include two pointers to ***PtrBase*** objects: a "Next" pointer, ***Next***, and a Previous pointer, ***Prev***. ***Next*** and ***Prev*** are used to maintain a bi-directional, doubly-linked ring of ***PtrBase*** objects whose derived pointers point to the same controlled object. ***Next*** and ***Prev*** are declared to be mutable, which allows them to be manipulated by the ***PtrBase*** even when the ***PtrBase*** is constant. Each ***PtrBase*** object points to the Next and Previous ***PtrBase*** object on the ring. In addition, the common base, ***PtrBase***, includes the member functions used to construct and destruct ring members and manipulate the ring. In particular, constructor and destructor member functions are provided in the common base along with the attach, detach, and single-member test member functions described above. Further, a member function, ***size_t Refs () const***, is provided for returning a value representing the number of members on a ring, which can be useful in debugging situations, for example.

[0025] The framework also comprises a class template for generating a different class of smart pointer for each class of object for which a smart pointer is used. The class template provides member functions to generate smart pointer classes specific to the respective classes of objects controlled by the smart pointers, in order to maintain type safety. In particular, the member function templates provided by the class function provide automatic conversion between smart pointers to objects of different classes in a common class hierarchy when such smart pointers point to different sub-objects of the same object.

[0026] The class template, ***template<typename T> class Ptr : public PtrBase***, is derived from ***PtrBase*** and generates a ***Ptr*** class specific to the type of the object being controlled. The sole attribute of a ***Ptr***, ***T * TPtr***, is the built-in pointer to the object. The template also provides a constructor, ***explicit Ptr (T * ptr)***, to set the built-in pointer to the value of ***ptr***. The constructor is made ***explicit*** to prevent an implicit conversion of a built-in pointer to a smart pointer.

[0027] The class template ***Ptr*** provides for automatic conversion of pointers to objects of different classes in a common class hierarchy when such pointers are pointing to different sub-objects of the same object. In particular, the template ***Ptr*** provides constructor and assignment member function templates that accept as arguments built-in pointers and smart pointers to objects of different classes, and uses such member function templates to automatically perform the necessary casting on the built-in pointers.

[0028] To this end, ***Ptr*** includes a member function template, ***template<typename TT> Ptr (const Ptr<TT> & other)***, to provide a copy constructor with automatic conversion of a smart pointer to one sub-object to a smart pointer to another sub-object of the same object. This is one of the cases for which the common base, ***PtrBase***, provides the ability for both smart pointers to belong to the same ring. This member function dynamically casts the ***other*** built-in pointer to ***this*** built-in pointer. Such a conversion will only be successful if the ***other*** built-in pointer and ***this*** built-in pointer point to objects of classes that are in the same class hierarchy and that are sub-objects of the same object. If the conversion is invalid, then ***this*** built-in pointer is assigned a value of zero and ***this PtrBase*** is initialized using the ***Reset*** member function. If, however, the conversion is valid, then ***this*** built-in pointer points to the same object as the ***other*** built-in pointer, and, accordingly, ***this PtrBase*** is attached to the ring associated with the ***other PtrBase***.

[0029] ***Ptr*** also includes a member function template, ***template<typename TT> Ptr & operator = (const Ptr<TT> & other)***, to provide a copy assignment with automatic conversion of a smart pointer to one sub-object to a smart pointer to another sub-object of the same object. This member function dynamically casts the ***other*** built-in pointer to ***this*** built-in pointer. If the ***other*** built-in pointer is different from ***this*** built-in pointer, then ***this PtrBase*** is tested to determine if it is the only ***PtrBase*** on its ring. If ***this PtrBase*** is the only ***PtrBase*** on its ring, then the object pointed to by ***this*** built-in pointer is deleted; otherwise, ***this PtrBase*** is detached from its ring. If the conversion is invalid, then ***this*** built-in pointer is assigned a value of zero and ***this PtrBase*** is initialized using the ***PtrBase::Reset*** member function. If, however, the conversion is valid, then ***this*** built-in pointer points to the same object as the ***other*** built-in pointer, and, accordingly, ***this PtrBase*** is attached to the ring associated with the ***other PtrBase***. In all cases, the member function concludes by returning a reference to ***this***.

[0030] Further non-template member functions are provided to enable smart pointer operations integrated with ring management. For example, a destructor member function, ~ ***Ptr ()***, is provided to delete the object to which a smart pointer points, when such smart pointer is the only member of its ring. If the smart pointer is not the only member of its ring, the smart pointer is detached from its ring. The destructor member function utilizes ***PtrBase::Is Only()*** and ***PtrBase::Detach()***. Similarly, a member function, ***void Delete ()***, is provided to delete the object to which a smart pointer points, when such smart pointer is the only member of its ring. Such member function operates in much the same way as the destructor member function, with the addition that the built-in pointer is set to zero.

[0031] A copy constructor member function, ***Ptr (const Ptr & other)***, is provided. The copy constructor member function copies the ***other*** built-in pointer to ***this*** built-in pointer. If ***this*** built-in pointer is zero, then ***this PtrBase*** is initialized using ***PtrBase::Reset()***. If ***this*** built-in pointer is not zero, then ***this PtrBase*** is attached to the ring associated with the ***other PtrBase*** using ***PtrBase::Attach()***.

[0032] Assignment of a built-in pointer to a smart pointer is provided by the member function ***Ptr & operator = (T * ptr)***. The assignment member function tests to see if the ***other*** built-in pointer is different from ***this*** pointer. If the two are different, the following steps are performed. First, before the assignment is made, ***this PtrBase*** is tested to see if it is the only ***PtrBase*** on its ring. If ***this PtrBase*** is the only ***PtrBase*** on its ring, then the object pointed to by ***this*** built-in pointer is deleted. If ***this PtrBase*** is not the only ***PtrBase*** on its ring, then ***this PtrBase*** is detached from its ring using the member function ***PtrBase::Detach()*** and is initialized using ***PtrBase::Reset()***. Second, ***this*** built-in pointer is assigned a copy of the ***other*** built-in pointer. Finally, in all cases, the member function returns a reference to ***this***.

[0033] A copy assignment member function, ***Ptr & operator = (const Ptr & other)***, is also provided. The copy assignment member function tests to see if the ***other*** built-in pointer is different from ***this*** built-in pointer. If the two are different, the following steps are performed. First, before the assignment is made, ***this PtrBase*** is tested to see if it is the only ***PtrBase*** on its ring. If ***this PtrBase*** is the only ***PtrBase*** on its ring, then the object pointed to by ***this*** built-in pointer is deleted. If ***this PtrBase*** is not the only ***PtrBase*** on its ring, then ***this PtrBase*** is detached from its ring using the member function ***PtrBase::Detach()***. Second, ***this*** built-in pointer is assigned a copy of the ***other*** built-in pointer. If ***this*** built-in pointer has a value of a zero, then ***this PtrBase*** is initialized using ***PtrBase::Reset***. If ***this*** built-in pointer does not have a value of zero, then ***this PtrBase*** is attached to the ring of the ***other PtrBase*** using ***PtrBase::Attach***. Finally, in all cases, the member function returns a reference to ***this***.

[0034] In addition, providing for the frequent need to mix the use of smart pointers in retrofit situations with built-in pointers, a member function, ***T * Remove ()***, is provided for removing a built-in pointer from the smart pointer environment. This member function converts a smart pointer to a built-in pointer without deleting the object to which the smart pointer points. First, the member function tests to determine if the smart pointer being removed is not the only pointer to its controlled object, which if true results in the member function returning zero. Otherwise, the member function returns a copy of the built-in pointer of the smart pointer and sets the built-in pointer of the smart pointer to zero.

[0035] The embodiment of the Example also enables using the smart pointer with C++ Standard Library collections by providing a member function, ***bool operator < (const Ptr & other) const***, defining a "less-than" condition which safely and correctly translates into the "less-than" condition for the controlled objects. If ***this*** built-in pointer is zero and the ***other*** built-in pointer is not zero, then the less-than member function returns "true". If ***this*** built-in pointer and the ***other*** built-in pointer are both not zero and the object to which ***this*** built-in pointer points is less than the object to which the ***other*** built-in pointer points, then the less-than member function returns "true". In all other cases, the less-than member function returns "false".

[0036] These and other advantages of the present invention will be apparent to those skilled in the art from the foregoing specification. Accordingly, it will be recognized by those skilled in the art that changes or modifications may be made to the above-described embodiments without departing from the broad inventive concepts of the invention. For example, while the embodiments are described in reference to memory-resident elements, the invention also has application to non- memory-resident elements such as IO mapped elements. It should therefore be understood that this invention is not limited to the particular embodiments described herein, but is intended to include all changes and modifications that are within the scope and spirit of the invention as set forth in the claims.

Example

```
//        mr_Ptr.hxx

#ifndef mr_Ptr_hxx
#define mr_Ptr_hxx

#include <cstddef>
#include <iosfwd>

namespace mr
{

        class PtrBase
        {
                mutable const PtrBase * Next, * Prev;
          protected:
                PtrBase ()
                {}
                ~PtrBase ()
                {}
                void Reset () const
                {
                        Next = Prev = this;
                }
                void Attach (const PtrBase & other) const
                {
                        Next = &other;
                        Prev = other.Prev;
                        Prev->Next = other.Prev = this;
                }
                void AttachX (const PtrBase & other) const
                {
                        other.Attach (*this);
                }
                void Detach () const
                {
                        Prev->Next = Next;
                        Next->Prev = Prev;
                }
          public:
                bool IsOnly () const
                {
                        return Next == this;
                }
                size_t Refs () const
                {
```

```
            size_t refs = 1;
            for (const PtrBase * link = Next; link != this; link = link->Next)
                ++refs;
            return refs;
        }
};

template<typename T> class Ptr : public PtrBase
{
        T * TPtr;
  public:
        Ptr ()
            : TPtr (0)
        {
            Reset ();
        }
        explicit Ptr (T * ptr)
            : TPtr (ptr)
        {
            Reset ();
        }
        template<typename TT> Ptr (const Ptr<TT> & other)
            : TPtr (dynamic_cast<T *> (static_cast<TT *> (other)))
        {
            if (TPtr == 0)
                Reset ();
            else
                Attach (other);
        }
        Ptr (const Ptr & other)
            : TPtr (other.TPtr)
        {
            if (TPtr == 0)
                Reset ();
            else
                Attach (other);
        }
        ~Ptr ()
        {
            if (IsOnly ())
                delete TPtr;
            else
                Detach ();
        }
        Ptr & operator = (T * ptr)
        {
            if (TPtr != ptr)
            {
                if (IsOnly ())
                        delete TPtr;
                else
                {
                        Detach ();
                        Reset ();
                }
                TPtr = ptr;
        }
        return *this;
}
template<typename TT> Ptr & operator = (const Ptr<TT> & other)
{
        T * tptr = dynamic_cast<T *> (static_cast<TT *> (other));
        if (TPtr != tptr)
        {
                if (IsOnly ())
                        delete TPtr;
                else
                        Detach ();
                if ((TPtr = tptr) == 0)
                        Reset ();
                else
                        Attach (other);
        }
        return *this;
}
Ptr & operator = (const Ptr & other)
{
        if (TPtr != other.TPtr)
        {
                if (IsOnly ())
                        delete TPtr;
                else
                        Detach ();
                TPtr = other.TPtr;
                if (TPtr == 0)
                        Reset ();
                else
                        Attach (other);
        }
        return *this;
}
void Delete ()
{
        if (IsOnly ())
                delete TPtr;
        else
```

```
            {
                Detach ();
                Reset ();
            }
            TPtr = 0;
        }
        T * Remove ()
        {
            if (!IsOnly ())
                return 0;
            T * tptr = TPtr;
            TPtr = 0;
            return tptr;
        }
        T * operator -> () const
        {
            return TPtr;
        }
        T & operator * () const
        {
            return *TPtr;
        }
        operator T * () const
        {
            return TPtr;
        }
        bool operator < (const Ptr & other) const
        {
            if (TPtr == other.TPtr)
                return false;
            if (TPtr != 0 && other.TPtr != 0)
                return *TPtr < *other.TPtr;
            return TPtr == 0;
        }
    };

}

#endif // mr_Ptr_hxx
```